



Distance Learning System

Obrada i generisanje izuzetaka

Object Oriented Programming Python

Greške u aplikaciji

- Greške u aplikacijama, mogu se podeliti na dva tipa:
 - **Sintaksne** (greška u sintaksi, lako se ispravlja)
 - **Logičke** (logička greška, teže se ispravlja)
- Takođe, greške se mogu podeliti po trenutku događanja:
 - **Greške u prevođenju**
 - **Greške u izvršavanju (runtime greške)**

Izuzeci

- Greške u izvršavanju, događaju se najčešće zbog nedostatka resursa, neispravnog indeksiranja, konverzije i slično
- Mi, kao developeri, ove greške moramo predvideti i sprečiti
- Python sadrži mehanizam koji omogućava da aplikacija na „miran“ način objavi kako nešto nije u redu, dajući developeru mogućnost da u tom trenutku reaguje. Ovaj mehanizam je sistem **izuzetaka**.
- Izuzetak je situacija u kojoj neki deo programa nije u stanju da funkcioniše po predviđenom planu, ali, za razliku od greške, dozvoljava programu da izvrši alternativu koja će sanirati problem i sprečiti prekid rada programa

Primer izuzetka (oopp-ex04 pythonexceptions.py)

- Ako bi pokušali da startujemo Python program:

```
x = 100 / 0
```

Izlaz bi bio sledeći:

```
ZeroDivisionError: division by zero
```

- Odnosno, Python bi nam dao do znanja da nešto nije u redu, emitovao grešku i prekinuo rad aplikacije
- Ali ovu grešku nije izazvala pogrešna aritmetička operacija, već **neobrađen izuzetak**

Tipovi obrade izuzetka

- U prethodnom primeru Python je prepoznala nepravilnost u vrednostima promenljivih, i na osnovu te nepravilnosti generisala **Exception** objekat, a zatim ga isporučila aplikaciji
- Exception objekat "isplivava" na površinu aplikacije u trenutku kada se pojavi u sistemu, i ako se niko ne "pozabavi" njime (obradi ga), isplivaće na samu površinu i aplikacija će prestati sa radom
- Za obradu izuzetka koristimo jednu od dve tehnologije:
 - Slanje izuzetka dalje
 - "Hvatanje" izuzetka

Hvatanje izuzetka

- Najčešća situacija obrade izuzetka je njegovo „hvatanje“
- Za hvatanje izuzetaka koriste se **try / except** blokovi
- Try except blokovi funkcionišu po sledećem principu:
 - Kreira se struktura, sa minimum dva bloka
try: ... except: ...
 - Deo koda u kome se očekuje izuzetak se smešta u **try** blok
 - Deo koji bi trebalo da se izvrši ukoliko ne uspe izvršavanje prvog bloka smešta se u **except** blok
 - Cela struktura, sintaksno izgleda ovako:

```
try:  
    y = 10 / 0  
except:  
    print("Hey, you can't divide by zero!")
```

Bočni efekti u try / except bloku

- Primer sa prethodnog slajda sprečava pojavljivanje greške, i ne prekida izvršavanje programa. Ipak, problem i dalje postoji, i u trenutku kada dođe do pokušaja deljenja sa nulom, kompletna grana programa će biti prekinuta (ako postoji finally block, on se izvršava i u ovom slučaju)

Nikada se
ne izvrši

```
y = 5
try:
    x = 100 / 0
    y = 10
except:
    print("Hey, you can't divide by zero!")
print(y)
```

U ovom trenutku
grana se prekida
A zatim se
ovde nastavlja

- Kolika će biti vrednost promenljive y nakon try / except bloka?

Vežba (oopp-ex04 add.py)

- Potrebno je kreirati program koji sabira brojeve, tako što konstantno traži od korisnika da unese prvi pa drugi operand
- Ukoliko korisnički unos nije u redu, program prikazuje poruku da promenljive nisu ispravne, a zatim nudi korisniku ponovni unos
- Program NE mora da nakon greške nastavi sa započetom operacijom, već treba da počne sledeću (unos oba operanda iznova)

Tipovi izuzetaka

- Nakon try bloka, sledi exception blok. Ovaj blok je parametrizovan za razliku od try bloka i kao parametar prihvata objekat klase očekivanog izuzetka:

```
except Exception:
```

- Ovo ne može biti bilo koja klasa već isključivo klasa **BaseException** ili klasa koja je nasleđuje (preporučuje se Exception kao generalni izuzetak)

```
except object:
```

- Takođe je važno da parametar catch bloka odgovara očekivanom izuzetku, inače on neće ni biti uhvaćen. Na primer, sledeći kod će izbaciti grešku:

```
try:
    x = 100 / 0
except IndexError:
    print("Hey, you can't divide by zero!")
```

Multiple except blokovi

- Videli smo da `IndexError` objekat nije ispravno reagovao na aritmetički izuzetak, ali da `Exception` objekat jeste. Ovo se događa zbog toga što se izuzeci hvataju po nivou specijalizacije.
- Klasa `Exception` je u samom vrhu svih korisnički definisanih klasa izuzetaka, i zbog toga „hvata“ sve izuzetke. Prilikom deljenja sa nulom, dolazi do specijalizovanog izuzetka: **`ZeroDivisionError`**, koji je čak dva koraka u hijerarhiji od **`Exception`** klase.
- **`IndexError`** je takođe specijalizovana izuzetak klasa, ali ne odgovara klasi `ZeroDivisionError`, i zato izuzetak nije adekvatno uhvaćen
- Python omogućava provere izuzetaka različitog tipa, pomoću multiple except blokova

```
try:
    x = 100 / 0
except IndexError:
    print("I will never be caught")
except ZeroDivisionError:
    print("Hey, you can't divide by zero!")
```

Multiple except blokovi

- Multiple except blokovi se obično formiraju tako da se izuzeci „hvataju“ od specijalnih ka generalnim. Dobra praksa je da se na kraju svih specijalnih slučajeva, konačno uhvati i Exception, koji će uhvatiti sve izuzetke koji su eventualno prošli filtraciju

```
try:
    x = [1,2,3]
    x[3] = 10
except IndexError:
    print("I will never be caught")
except ZeroDivisionError:
    print("Hey, you can't divide by zero!")
except Exception:
    print("I am here just in case")
```

Finally blok

- Osim try i except, postoji još jedan, opcioni blok unutar try except strukture. To je blok **Finally**
- Finally blok se izvršava u bilo kom slučaju. Gotovo isto kao da smo napisali kod izvan kompletnog try except bloka (ali ne i potpuno isto)
- Ovaj blok koristi se najčešće za raspuštanje resursa
- Finally blok nema parametre

```
try:
    x = [1,2,3]
    x[3] = 10
except IndexError:
    print("I will never be caught")
except ZeroDivisionError:
    print("Hey, you can't divide by zero!")
except Exception:
    print("I am here just in case")
finally:
    print("I will be executed anyway")
```

Ručno izbacivanje izuzetaka

(oopp-ex04 manualexceptionthrow.py)

- Izuzetak možemo izbaciti i ručno, u bilo kom trenutku izvršavanja programa
- Izuzetak se ručno može izbaciti naredbom **raise**.
- Naredbi raise mora uslediti validan Exception objekat

```
ex = Exception("My Exception!")  
raise ex
```

ili samo

```
raise Exception("My Exception!")
```

Ručno izbacivanje izuzetaka

- U primeru, kreirana je funkcija za deljenje brojeva do deset
- Ova funkcija nema problem sa nulom kao deliocem, tada jednostavno vraća kao rezultat broj 0, ali ima problem sa operandima koji su veći od 10
- Pošto Python nema problem sa deljenjem brojeva većih od deset, neće izbaciti nikakav izuzetak ukoliko ubacimo takav broj u metod, pa zato moramo ovu situaciju obraditi ručno

```
def divide(a,b):  
    if b==0:  
        return 0  
    if a>10 or b>10:  
        raise ArithmeticError("Larger than 10")  
    else:  
        return a+b  
  
print(divide(14,1))
```

ArithmeticError: Larger than 10

Struktura klase Exception

- Pojavljivanje izuzetka ukazuje na to da je došlo do nepravilnosti u programu. Ali takođe, objekat klase izuzetak nosi u sebi dodatne informacije vezane za izazvanu nepravilnost
- Ove informacije možemo dobiti putem svojstava izuzetka ili modula **traceback**:

```
import traceback
```

```
except Exception as ex:  
    print(ex.__cause__)  
    print(ex.__class__)  
    print(traceback.format_exc())
```

Korisnički definisani izuzeci

(oopp-ex04 customexceptions.py)

- Možemo kreirati sopstveni izuzetak, dovoljno je da mu damo ime, i nasledimo klasu Exception

```
class MyException(Exception):  
    pass
```

- Ukoliko postoji potreba, ponašanje klase može se modifikovati prepisivanjem metoda

```
class MyException(Exception):  
    def __str__(self):  
        return "Something nice"
```


Slanje izuzetka dalje

- Ako ne želimo da obrađujemo izuzetak u metodi - ne moramo, ali to će imati posledice na deo programa koji je aktivirao metodu

```
def throwingEx():  
    raise MyException()  
  
def throwingEx1():  
    throwingEx()  
  
try:  
    throwingEx1()  
except MyException as ex:  
    print("Error in function")
```

Vežba (oopp-ex04 calculator)

- Postojeću aplikaciju potrebno je obezbediti tako da ne prijavljuje grešku

```
class Calculator:
    @staticmethod
    def calculate(a, b, op):
        if op == "+":
            return a + b
        if op == "-":
            return a - b
        if op == "/":
            return a / b
        if op == "*":
            return a * b
        return 0

x = Calculator.calculate(5,0,"/")
print(x)
```

Vežba (oopp-ex04 user)

Postoji sledeća klasa User:

```
class User():
    def __init__(self, id, firstName, lastName, email):
        self.id = id
        self.firstName = firstName
        self.lastName = lastName
        self.email = email

u = User(10, "Petar", "Jackson", "peters@mail.mm")
print(u)
```

Potrebno je kreirati klasne izuzetke za nepravilan unos ID-a, imena, prezimena i E-mail-a

Potrebno je implementirati sistem provere u konstruktor klase User tako da ukoliko je ID veći od 100, bude izbačen InvalidIdException, ako su firstName, lastName i E-mail polja prazna, bude izbačen InvalidFirstNameException, InvalidLastNameException ili InvalidEmailException

Potrebno je instancirati ovu klasu u Main projektu

Logging

<https://docs.python.org/3.8/library/logging.html>

- Logovanje omogućava precizno praćenje i prikazivanje različitih vrsta objava koje program šalje tokom izvršavanja
- Za razliku od standardnog ispisa, log se ispisuje filtrirano, po nivoima i moguće je usmeriti ispis na različite izlaze
- Nivoi logovanja su: **DEBUG**, **INFO**, **WARNING**, **ERROR** i **CRITICAL**
- Emitovanje poruka određenog nivoa, vrši se istoimenim funkcijama:
 - `logging.info("Ah, what a nice coffie, let me get some cold watermelon")`
 - `logging.debug("I ate 5 pieces of watermelon, and lost 0.0034 seconds")`
 - `logging.warning("Woopsie... there is something in my belly?")`
 - `logging.error("Huh...office...fast!")`
 - `logging.critical("Too late. It's all over")`

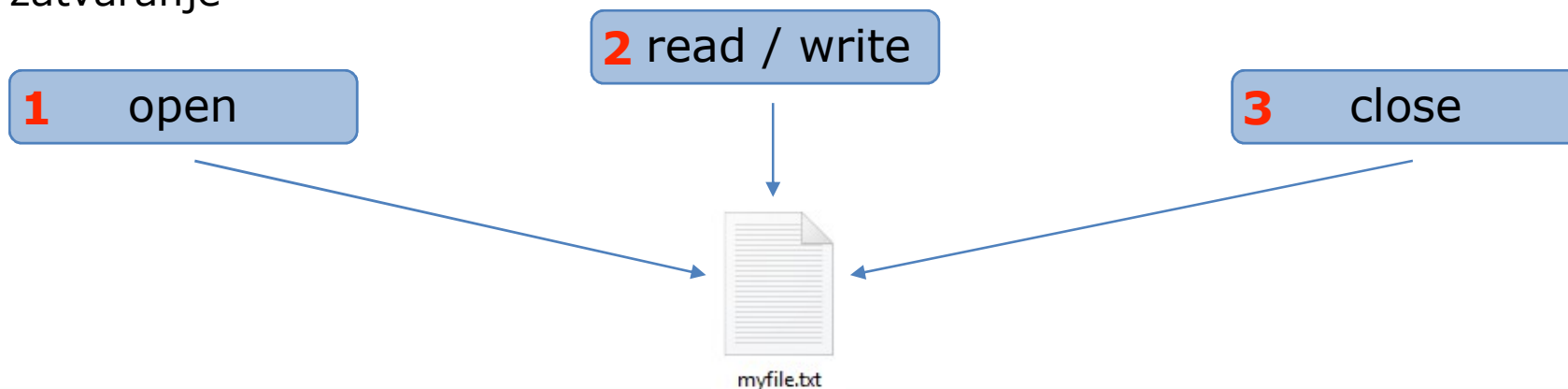
Konfigurisanje log-a

- Da bi logger ispisivao poruke određenog nivoa, potrebno ga je konfigurisati
- Logger se konfiguriše na početku programa, pomoću config funkcija
- config funkcije su **logging.basicConfig**, ili `logging.config.fileConfig` i `logging.config.dictConfig`

```
logging.basicConfig(  
    format='%(asctime)s:%(levelname)s:%(message)s',  
    level=logging.INFO)
```

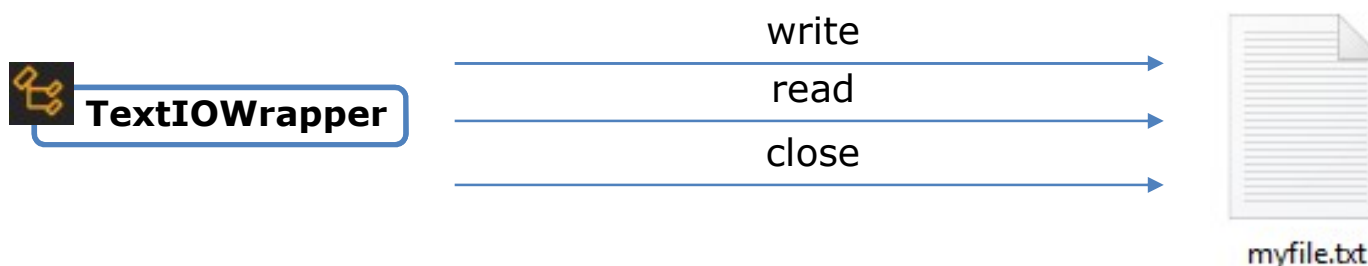
Rad sa fajlovima

- Kada aplikacije žele da trajno sačuvaju neke podatke ili da ih učitaju, u tu svrhu koriste fajlove
- Sa fajlovima se radi i implicitno prilikom korišćenja baze podataka ili mreže
- Fajlove možemo koristiti baferovano (čitamo sve odjednom) ili nebaferovano (čitamo deo po deo)
- Najčešće, procedura rada sa fajlovima podrazumeva otvaranje, manipulaciju i zatvaranje



Otvaranje fajla i Klasa TextIOWrapper

- Funkcije za rad sa fajlovima nalaze se u modulu **io**
- Neke od ovih funkcija takođe imaju i ugrađene sinonime (tako se funkcija **io.open**, može startovati i samo kao **open**)
- Funkcija **open** otvara fajl i vraća objekat klase **TextIOWrapper** koji sadrži metode za upravljanje tokom podataka fajla



- **TextIOWrapper** je baferovana verzija klase **TextIOBase** i ima mogućnost čitanja i pisanja teksta u tok podataka
- Fajl zatvaramo zatvaranjem strima, metodom **close**

Otvaranje i zatvaranje fajla

- Fajl otvaramo funkcijom **open** (ili metodom `io.open`)
- Metoda **read** preuzima kompletan sadržaj fajla u tekstualnom formatu
- Metoda **close** zatvara fajl

```
f = open("myfile.txt")  
print(f.read())  
f.close()
```

- U primeru, fajl se otvara u podrazumevanom režimu (čitanje)
- Metod `read` se poziva sa podrazumevanom dužinom (kompletan fajl)

Režimi rukovanja fajlom

- Prilikom otvaranja fajla, treba proslediti režim pod kojim hoćemo da upravljamo fajlom
- Bitniji režimi su: čitanje (**r**), upis (**w**), dodavanje (**a**), ali osim njih, postoje i mnogi drugi režimi (rb - binarno čitanje, r+ - čitanje i upis, rb+ binarno čitanje i upis, wb - binarni upis, w+ - upis i čitanje, wb+ - binarni upis i čitanje, ab - binarno dodavanje, a+ - dodavanje i čitanje, ab+ - binarno dodavanje i čitanje)
- Za eksplicitno određivanje režima rada sa fajlom, dodajemo parametar funkciji open

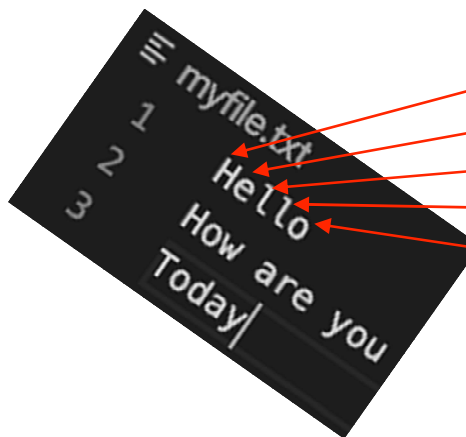
```
f = open("myfile.txt","r")  
print(f"This file is in: {f.mode} mode")  
f.close()
```

→ This file is in: r mode

Ponašanja pod različitim režimima

- U režimu čitanja (**r**), postojeći fajl se otvara i možemo ga iščitavati
- Čitanje se vrši metodom `read` ili njenim derivatima
- `Read` prihvata broj bajtova koji će biti pročitani
- Fajl mora postojati inače će doći do greške

- Ovaj režim ima varijacije: **rb**, **r+** i **rb+**



```
f = open("myfile.txt", "r")  
  
print(f.read(1))  
print(f.read(1))  
print(f.read(1))  
print(f.read(1))  
print(f.read(1))  
  
f.close()
```

Režim upisa - w

- Kod režima upisa, ekskluzivno se upisuju podaci u fajl
- Fajl se uvek kreira iznova i njegov postojeći sadržaj biva obrisano
- Ovaj režim ima varijacije: **wb**, **w+** i **wb+**

```
≡ myfile.txt  
1 Hello  
2 How are you  
3 Today|
```

```
f = open("myfile.txt", "w")  
f.write("Hello world")  
f.close()
```

```
≡ myfile.txt  
1
```

```
≡ myfile.txt  
1 Hello world
```


Režim dodavanja - a

- Ova režim je sličan režimu upisa, osim što se postojeći fajl ne briše već se marker postavlja na njegov kraj
- Ukoliko fajl ne postoji, biva kreiran
- Ovaj režim ima varijacije: **ab**, **a+** i **ab+**

```
f = open("myfile.txt","a")  
f.write("Really...")  
f.close()
```

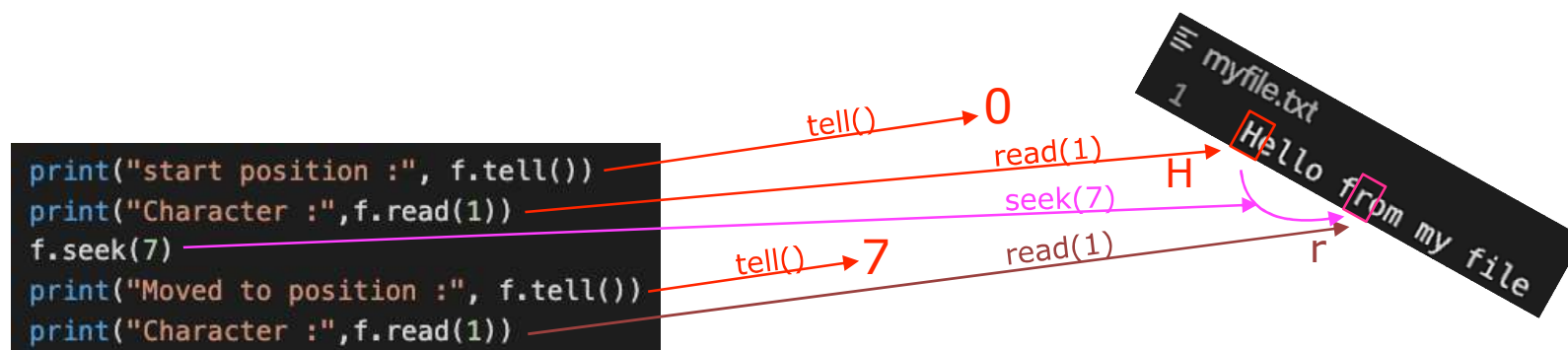
```
≡ myfile.txt  
1 Hello  
2 How are you  
3 Today|
```

```
≡ myfile.txt  
1 Hello  
2 How are you  
3 TodayReally...
```



Manipulacija sadržajem fajla

- Najčešće želimo da nešto upišemo u fajl, ili da nešto iz njega pročitamo
- Metode **read** i **write** respektivno čitaju ili upisuju sadržaj u otvoreni fajl
- Bitan faktor je marker fajla. Marker predstavlja trenutnu poziciju od koje počinje upis ili čitanje
- Poziciju markera možemo programabilno saznati ili korigovati
- Metode za pomeranje markera su **seek** (pomera marker na određenu poziciju, napred ili nazad) i **tell** (vraća aktuelnu poziciju markera)



Manipulacija fajlovima

- Najčešće operacije nad fajlovima su brisanje, kopiranje, premeštanje i slično. Sve ove operacije dostupne su u modulima **os** i **shutil**

Brisanje datoteke	→	<code>os.unlink("myfile.txt")</code>
Brisanje direktorijuma	→	<code>os.rmdir("mydir")</code>
Rekurzivno brisanje direktorijuma	→	<code>shutil.rmtree("mydir")</code>
Kopiranje datoteke	→	<code>shutil.copyfile("myfile.txt", "yourfile.txt")</code>
Izmeštanje datoteke	→	<code>shutil.move("yourfile.txt", "theirfile.txt")</code>

Izlistavanje direktorijuma

- Najčešće operacije nad fajlovima su brisanje, kopiranje, premeštanje i slično. Sve ove operacije dostupne su u modulima **os** i **shutil**
- Iako s

```
for entry in os.walk("mydir"):
    print(entry)
```

Vežba - authentication

(oopp-ex04 auth)

- U fajlu se nalazi podaci o korisnicima u sledećem formatu:

peter,123,150

sally,234,220

john,hello,350

...

- Potrebno je kreirati program u kome će korisnik unositi korisničko ime i šifru a zatim moći da vidi svoje stanje na računu

```
Username: peter
Password:
Hello peter, your current balance is $150
```

Napomena: Pokušaj da istražiš i interesantne module: csv i getpass

Vežba - kasa

(oopp-ex04 kasa)

- U fajlu se nalaze podaci o proizvodima, u sledećem formatu:

1,Deterdžent,120,50

2,Čokoladna bananica,12,200

3,Čvarci,400,15

...

- Potrebno je kreirati program u kome korisnik (prodavac) može da:
 - Izlista proizvode
 - Ubaci proizvode u korpu
 - Dobije izračunatu cenu korpe
 - Izvrši prodaju (smanjuju se količine prodatih proizvoda)

Zadatak - Quiz

- U tekstualnom fajlu se nalaze pitanja za kviz
- Za svako pitanje, postoje četiri ponuđena odgovora i broj bodova
- Korisnik, unosi svoje ime u program i dobija pet odabranih pitanja (preuzetih iz liste pitanja)
- Korisnik odgovara na pitanje odabirom broja odgovora
- Po završetku
 - izračunava se broj tačnih odgovora i rezultat korisnika se čuva u istom fajlu u kome se nalaze i pitanja
 - Prikazuje se lista imena i rezultata, za 10 korisnika sa najboljim rezultatima

Vežba - prevođenje

(oopp-ex04 simplecompiler)

- Korisnik prilikom startovanja programa unosi naziv izvornog fajla napisanog jednostavnom sintaksom
- Sintaksa podrazumeva komandu echo (štampanje na izlaz) i aritmetičku operaciju
- Potrebno je napraviti program koji će opisani izvorni fajl prevesti u programski jezik python

program.php

```
echo 'hello man\n';  
echo '2 + 3 = '  
echo 2+3;  
echo '\n';
```



program.py

```
print('hello man\n',end='')  
print('2 + 3 = ',end='')  
print(5,end='')  
print('\n',end='')
```

Testiranje Python aplikacije

- Testiranje aplikacije, ne razlikuje se mnogo od testiranja u ostalim tehnologijama, i može se podeliti na:
 - Testiranje performansi
 - Load
 - Stress
 - Unit testiranje
 - Testiranje interfejsa
 - Testiranje ponašanja
 - Testiranje stanja
 - ...
- U nastavku će biti obrađeno unit testiranje u Python-u

Šta je unit testiranje

- Unit testiranje je testiranje klasa i metoda (pre svega metoda)
- U unit testu, proveravamo da li metode imaju očekivano ponašanje tako što ih startujemo, a zatim proverimo da li su na osnovu unetih parametara, vratili očekivane vrednosti

```
def hello(a,b):  
    return a + b  
  
expected = 5  
p1 = 2  
p2 = 3  
res = hello(p1,p2)  
if res == expected:  
    print("Test passed")  
else:  
    print("Test failed")
```

Zašto vršimo unit testiranje

- Unit testiranje je važno najviše zbog toga da ne bismo novim funkcionalnostima projekta poremetili postojeće funkcionalnosti
- Unit testiranje treba sprovoditi nakon izmena u kodu
- Unit testovi bi trebalo da budu u sastavu projekta, ali dovoljno odvojeni da možemo da ih po potrebi isključimo iz procesa prevođenja
- Za unit testiranje, najčešće se koriste gotova rešenja, radije nego ručno pisana
- U Python-u, za unit testiranje se može koristiti modul **unittest**.

Unit testiranje klase (oopp-ex04 calculatortest.py)

- Recimo da postoji klasa Calculator, i čiji metod add želimo da podvrgnemo unit testiranju

```
import unittest
class Calculator:
    def add(self,a,b):
        return a + b
```

Unit testiranje klase Calculator

<https://docs.python.org/3/library/unittest.html>

- Da bi klasa bila Unit test klasa, treba da nasledi klasu **unittest.TestCase**
- Naziv svih metoda test klase, mora počinjati sa test, da bi bile uzete u obzir za testiranje
- U svakoj test metodi, proverava se vrši pozivima assert metoda klase TestCase
- Procedura testiranja počinje pozivom funkcije unittest.main() ili konzolno

```
class TestCalculator(unittest.TestCase):  
    def test_add(self):  
        p1 = 2  
        p2 = 3  
        exp_result = 5  
        instance = Calculator()  
        result = instance.add(p1,p2)  
        self.assertEqual(result,exp_result)
```

```
python3 -m unittest calculatortest.py
```

```
unittest.main()
```


JUnit unit testiranje

- U zavisnosti od načina startovanja, test će prikazati eventualne greške

```
bash-3.2$ python3 -m unittest calculortest.py
.  
-----  
Ran 1 test in 0.000s  
OK
```

OK

GREŠKA

```
bash-3.2$ python3 -m unittest calculortest.py
F  
-----  
FAIL: test_add (calculortest.TestCalculator)  
-----  
Traceback (most recent call last):  
  File "calculortest.py"  
    , line 13, in test_add  
      self.assertEqual(result,exp_result)  
AssertionError: 5 != 6  
-----  
Ran 1 test in 0.000s  
FAILED (failures=1)
```

JUnit assertion metode

- Assert metode su metode za proveru određenog stanja. Do sada smo koristili metod **assertEqual**, koji proverava da li jedna vrednost odgovara drugoj. Ali takođe, postoje i druge assert metode:
assertEqual, assertNotEqual, assertTrue, assertFalse, assertIs, assertIsNot, assertIsNone, assertIsNotNone, assertIn, assertNotIn, assertIsInstance, assertNotIsInstance